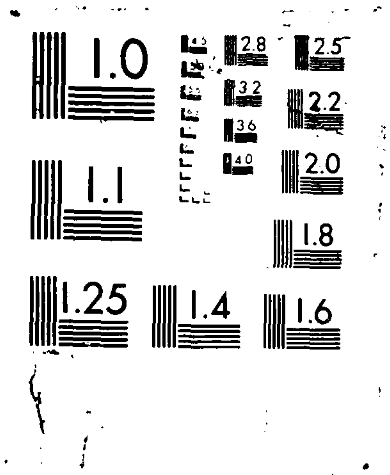

1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0

(4)

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER none	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Programming Solutions to the Algorithm Contraction Problem		5. TYPE OF REPORT & PERIOD COVERED Technical Report
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Philip A. Nelson and Lawrence Snyder		8. CONTRACT OR GRANT NUMBER(s) N00014-85-K-0328
9. PERFORMING ORGANIZATION NAME AND ADDRESS University of Washington Department of Computer Science Seattle, Washington 98195		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research Information Systems Program Arlington, VA 22217		12. REPORT DATE April 1986
		13. NUMBER OF PAGES 4
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this report is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Algorithm contraction problem: parallel programming

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Algorithms for the parallel solution of problems are usually designed assuming an unlimited number of processors. Physical parallel machines have a fixed number of processors. The algorithm contraction problem arises when an algorithm requires more processors than are available on the physical machine. We present tools for comparing algorithm contractions based on bottle neck communication paths. We apply these tools to minimum, matrix product and sorting.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

PROGRAMMING SOLUTIONS TO THE ALGORITHM CONTRACTION PROBLEM

Philip A. Nelson, Lawrence Snyder
Computer Science Department, FR35
University of Washington
Seattle, Washington 98195

Abstract

Algorithms for the parallel solution of problems are usually designed assuming an unlimited number of processors. Physical parallel machines have a fixed number of processors. The algorithm contraction problem arises when an algorithm requires more processors than are available on the physical machine. We present tools for comparing algorithm contractions based on bottle neck communication paths. We apply these tools to minimum, matrix product and sorting.

Introduction

Algorithms for parallel computers are usually designed assuming an unlimited number of processors. For non-shared memory parallel algorithms, this assumption generally manifests itself by the algorithm utilizing one processor "per point", or some other input size-dependent processor allocation. The physical machine has only a fixed number of processors, of course, which will almost certainly be less than the number required by the algorithm. In order to make the logical processes of the algorithm conform to the physical processors of the machine, we must group processes together into a module to be executed on a single physical machine. This activity is called *contraction*[13]. The way this contraction is performed can have a significant affect on performance.

Consider two examples based on an grid $n \times n$ of processes, *i.e.* the processes communicate with their four nearest neighbors:

(1) There is much process-to-process communication and approximately equal computation required of each process.

(2) There is little process-to-process communication and the amount of computation per process is proportional to its j index, *e.g.* process i,j iterates j times.

Suppose we have only one fourth the required number of processors and now compare two ways of forming contractions of four processes per processor[4]: *Coalescing* groups of adjacent 2x2 subarrays; *folding* groups as if the grid is folded in half and then in half again, *i.e.* i,j $(1 \le i,j \le \frac{n}{2})$ is associated with $i,n-j+1$, $n-i+1,j$ and $n-i+1,n-j+1$. Clearly, algorithm (1) should be contracted by coalescing because the process-to-process communication for the processes sharing the same processor will become intraprocessor communications (*i.e.* fast memory references) rather than slow interprocessor communication; folding would not be as attractive because no communication is saved by locality. Alternatively, algorithm (2) should be contracted by folding because the work is balanced since each processor will perform a matching amount of long and short computations; coalescing would not be as attractive because the processors receiving processes with large indexes will become a bottleneck.

Using the results of Berman and her colleagues[3], an algorithm can be be automatically contracted, and this seems to be the best approach when nothing is known about the algorithm. At the other end of the spectrum, however, the programmer has "complete" knowledge about the algorithm. How should he be guided when performing his own contraction? In this paper we develop some apparatus to guide the programmer who must contract a algorithm. We will provide some case studies of contraction that show an unexpected diversity and we offer some general contraction strategies that can find application in other algorithms. Contraction is a nontrivial problem for parallel programmers[13], and so a secondary goal here is to expose it as an important topic for study and a subject suitable for rigorous analysis.

Definitions

The generic parallel architecture under consideration in this paper is a non-shared memory model. It is a collection of homogeneous sequential computers operating asynchronously and connected in a communication network that is a bounded degree graph[13]. A single "edge" in the graph provides bidirectional communication between two processors. The CHiP[11] architecture is an example of this generic architecture.

Supported in part by National Science Foundation Grant DCR-8416878 and by Office of Naval Research Contract No. N00014-85-K-0328.

The method used for programming this model consists of defining a sequential program for each processor and a communication graph. We are assuming a configurable architecture. (The problem of mapping a communications graph onto a different processor connection graph is discussed by Berman and Snyder[4] and Bokhari[5].) Communication is explicitly shown in the sequential code by specifying a data value to be sent to the processor connected by a given edge.

The algorithm contraction problem arises when an algorithm that is designed for use on n processors must be mapped to a physical parallel computer with only $p<n$ processors. The programmer must decide which logical processes are to be mapped to the physical processors. Assuming that the logical processes have balanced loads (they run for the same length of time), we would like the physical processors to have balanced loads. This is done by mapping the same number of logical processes to each physical processor. The number of logical processes assigned to two arbitrary physical processors should differ by at most an additive constant c. For most contractions, it would be best to have $c=1$.

The contraction induces a communication graph for the p physical processors. This new graph is defined by logical processes needing to communicate with other logical processes not mapped to the same physical processor. We assume that if a logical process in processor i needs to communicate with a logical process in processor j, there is a physical edge connecting the two processors in the new graph. The contraction may map many of these logical edges to one physical edge in the new graph. That is, we are allowing only one edge between physical processors. Under the assumption of a bounded degree graph for the generic architecture, this induced graph must also be of bounded degree.

As an example of contraction, let us assume we have an algorithm with a tree graph. Consider the contraction to 5 processors shown in Figure 1. This contraction caused an increase in the degree; for example, the new root vertex has four descendants. Using this kind of a contraction, it can be shown that given p processors, contracting an algorithm with at least p^2 logical processes requires degree $p-1$. Figure 2a shows a contraction of the tree to 4 processors. An extension of this method yields a binary tree in the p processors.

Figure 2b gives another contraction to 4 processors. This contraction is derived by the recursive tree construction given by Leiserson[9]. Given two instances of a tree each with an associated free node, we can build a new tree and an associated free node. This produces a linear area layout in the plane with several desirable properties, one of which is the constant number of external edges.

In this paper, we will be considering the contribution of the communication time to the performance of the contracted algorithm. Unless otherwise stated, we assume that a communication between processing elements costs a fixed time t_c. During this time, no other communication in

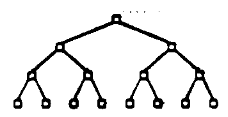
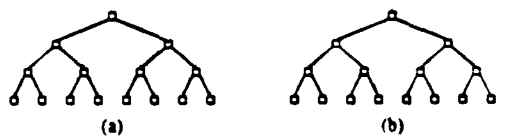


Figure 1: A 5 processor contraction.



(a) (b)

Figure 2: Two 4 processor contractions.

88 4 1 068

the same direction may take place. We are specifically allowing all edges to have simultaneous communication. Communication internal to a processor costs the fixed time t_i. We also assume that $t_c \gg t_i$.

We would like to develop tools for reasoning about the relative merits of different contractions. This includes their communication costs and their execution times. To aid in this objective we give the following definitions.

Let $A = (V, E)$ be an algorithm where V is a set of logical processes (vertices and associated programs) and $|V| = n$, E is a set of edges (V_1, V_2), $V_1, V_2 \in V$.

Let $M(A, p) = B$ be a contraction of algorithm A into algorithm B where B uses p processors and $p < |V_A|$. The contraction M maps elements of V_A onto V_B such that the number of elements of V_A mapped to an arbitrary element of V_B differs by no more one from the number of elements of V_A mapped to any other element of V_B.

Let $w(e)$, the weight of e, for $e = (V_1, V_2)$, be the larger of the number of messages from V_1 to V_2 and the number of messages from V_2 to V_1.

Let $K(A) = \underset{e}{MAX}\, w(e)$, for $e \in E$, be the communication "cost" of A. This cost is an estimate of the minimum communication time required for the algorithm. Due to dependancies, the actual communication cost may be more.

Let $T(A)$ be the execution time for A.

PROPOSITION 1: For a given A, p, M_1, and M_2, and $t_c > t_i$, if $K(M_1(A, p)) < K(M_2(A, p))$ then $T(M_1(A, p)) \le T(M_2(A, p))$.

This proposition is formalizing the notion that the bottleneck edge will be a lower bound on the time required for the execution of the mapped algorithm. If the processors have a small amount of computation relative to the communication, the execution time will depend on the communication time. The bottleneck edge of the contraction M_1 will require a minimum of $t_c K(M_1(A, p))$ time, which is less than $t_c K(M_2(A, p))$. With a higher minimum communication time, we can not expect M_2 to execute in less time than M_1. If the processors have a large amount of computation in ratio to the communication, the computation time will dominate, yielding near equal times. Even in this case, M_1 uses less time for communication than M_2. This proposition then motivates us to map the busiest edges of an algorithm to internal edges.

Case Studies

We now look at several parallel algorithms and some contractions. We approach these by considering algorithms with similar communication graphs. The three graphs considered are the tree, grid, and binary n-cube.

Tree algorithms

There are several algorithms that run on complete binary trees (Figure 1) having similar characteristics, like the aggregation operations of minimum and global sum. All processors have a value and we want to compute a global value that depends on all these values. Leaf processors send their value to their parents. Internal processors take the minimum (sums) of their own value and their children's values and then send the result to their parents. The final value will be computed at the root processor in $O(\log n)$ time. The communication in these algorithms requires one message over each edge for each global minimum (sum). For a single minimum we have $K(\mathit{minimum}) = 1$.

Consider the contraction in Figure 2a. Let us call this contraction $M_1(\mathit{minimum}, p)$. Each edge in the original algorithm requires one message. Each edge in the smaller graph has 4 edges from the original graph. Since we have only one connection between the physical processors, we have 4 messages for each edge. For an arbitrary n (size of original algorithm) and p (the number of processors) we have $K(M_1(\mathit{minimum}, p)) = \frac{n}{p}$.

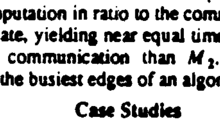


Figure 3: Berman and Snyder tree contraction.

A similar contraction to Figure 2a is touched on by Berman and Snyder[4]. Figure 3 shows this contraction. This is achieved by "folding" the tree. As Berman and Snyder notice, this contraction, M_2, has $K(M_2(\mathit{minimum}, p)) = \frac{n}{p}$.

Consider the contraction in Figure 2b. Let us call this contraction $M_3(\mathit{minimum}, p)$. We note that each edge in the smaller graph has at most one edge from the original graph in each direction. For an arbitrary n and p we have $K(M_3(\mathit{minimum}, p)) = 1$.

Proposition 1 tells us that since M_3 has a smaller K, it is the preferrable contraction. Both M_1 and M_2 depend on n and p for their cost. But, M_3 has a constant cost, regardless of n and p. In fact, this contraction is optimum for all tree algorithms that have identical edge weights and unidirectional communication (all toward the root or all toward the leaves).

We first look for a lower bound. Since the tree is connected, the physical processors must be connected. This requires at least one incident edge for each physical processor. The smallest cost $K(M(A, p))$ would be where a maximum of one logical edge was mapped to a physical edge. Therefore, $K(M(A, p)) \ge K(A)$, the cost of the original algorithm.

LEMMA 2: For complete binary tree algorithms with balanced processor loads, equal edge weights, and unidirectional communication, algorithm contraction based on Leiserson's binary tree layout technique yields optimum results.

PROOF: For the mapping $M_3(A, p)$, each processor contains a complete subtree and an "extra" node. The extra nodes are used in the tree above the subtrees contained in the processors. Therefore, there at most 4 external connections. Of these four, two edges are used to receive(send) data from(to) the children of the extra node, and two edges are used to send(receive) data to(from) the subtree's and the extra node's parents. Since the root of the subtree and the extra node are not at the same level in the tree, edges with data flowing in the same direction can not be connected to the same physical processor. (It is possible to have two of these edges over the same physical edge, but the data moves in opposite directions.) This gives the same weight to the physical edges as the original edges. Therefore, $K(M_3(A, p)) = K(A)$, which is the lower bound. $\square$

Notice that this layout technique will place two logical edges in the same physicial edge for some physical edge. For tree algorithms with bidirectional communication, we then get $K(M_3(A, p)) = 2K(A)$.

To help verify these results, the minimum algorithm was programmed using the Poker parallel programming environment[12]. Both M_1 and M_3 were programmed. Each contraction was timed using 4 and 16 data items per processor with 4 and 16 processors. The results of these timings are given in Table 1. Each "tick" represents a mircosecond on the 64 processor Pringle.

Grid algorithms

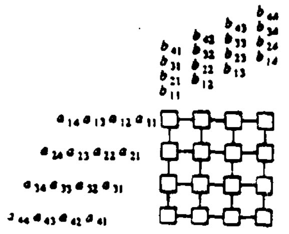
We next look at algorithms that run on a grid interconnection. Consider the matrix product algorithm for the Wavefront Array Processor(WAP)[8]. It uses n^2 processors for the $n \times n$ matrix product $AB = C$. The data is fed in along the top n processors and from the left n processors. The matrix A is arranged to enter column by column, starting with the first column. The matrix B is arranged to enter row by row, starting with the first row. (See Figure 4.) All processors execute identical procedures. The result, c_{ij}, is initialized to zero. A loop is executed n times that reads an A value from the left and a B value from above, multiplies them together, and adds the result to c_{ij}. The A and B values are sent to the right and down, respectively. This causes the upper left processor to be the first processor to start execution. As the data moves into the array, there is a wavefront of executing processors on the cross diagonal. Each edge is used to send all of one row of A or one column of B. For the WAP algorithm we have $K(WAP) = n$.

Consider the contraction in Figure 5. Let us call this contraction $M_1(WAP, p)$. This is the contraction done by cutting the graph into p equal size connected subgraphs and assigning one process from each subgraph selected from corresponding positions to a single processor. The

Minimum: ticks for n (items) on p (processors)				
Contraction	16 on 4	64 on 16	64 on 4	256 on 16
M_1	11650	20568	33496	105801
M_3	4356	7682	8878	12067

Table 1: Timings of the minimum algorithm.

A-1



b_{44} b_{34} b_{24} b_{14}
b_{43} b_{33} b_{23} b_{13}
b_{42} b_{32} b_{22} b_{12}
b_{41} b_{31} b_{21} b_{11}
a_{14} a_{13} a_{12} a_{11}
a_{24} a_{23} a_{22} a_{21}
a_{34} a_{33} a_{32} a_{31}
a_{44} a_{43} a_{42} a_{41}

Figure 4: WAP organization

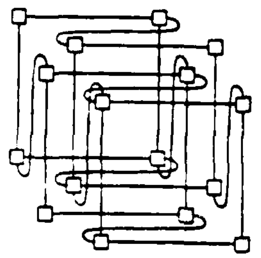


Figure 5: A contraction of 16 logical processes to 4 processors.

physical connection graph, shown in Figure 5, is a grid with end around (i.e. toroidal) connections. For each logical process in a physical processor, there are horizontal and vertical communication paths. Since we have $\frac{n^2}{p}$ logical processes in a processor, the number of logical edges using one processor-to-processor connection is $\frac{n^2}{p}$. Since all horizontal and vertical edges have the same number of messages, n, we have $K(M_1(WAP,p)) = \frac{n^3}{p}$.

Consider the contraction in Figure 6. Let us call this contraction $M_2(WAP,p)$. This is the contraction done by cutting the graph into p equal size connected subgraphs and assigning an entire subgraph to a processor. We see that only the perimeter processes have edges that go from processor-to-processor. Also, notice that no end around connections are needed. The number of communication paths over one processor-to-processor connection is $\sqrt{\frac{n^2}{p}}$. Each communication path requires n messages giving $K(M_2(WAP,p)) = \frac{n^2}{\sqrt{p}}$.

Comparing the two contractions, we see that $K(M_2(WAP,p))$ is smaller than $K(M_1(WAP,p))$ by a factor of $\frac{n}{\sqrt{p}}$. Proposition 1 tells us that M_2 is the better contraction. We conjecture that M_2 is the best contraction that can be achieved for grid algorithms. The basis for this conjecture is that this contraction has the smallest perimeter for a given area, and has been commonly used for contraction in published algorithms, for example for the Jacobi iterative method[1] and for the conjugate gradient method[6].

Both M_1 and M_2 were programmed using Poker. Table 2 summarizes the results of the timings. As predicted, M_2 was the faster contraction, but because the communication time is not the only time consuming part in these algorithms the difference is perhaps not as dramatic as might be seen on a larger problem.

Binary n-cube algorithms

We now look at two algorithms for the binary n-cube. The first algorithm is the divide-and-conquer algorithm for matrix product given by Nelson[10]. The other algorithm is Batcher's bitonic sorting algorithm[2].

The matrix product algorithm takes two $n \times n$ matrices, A, and B, and computes their product $C = AB$. A and B are assumed to be in row major order in the binary n-cube of order $2k$, where $k = \log n$. The algorithm views A and B as a 2×2 matrix of $\frac{n}{2} \times \frac{n}{2}$ matrices. The 2×2 matrix algorithm is then used to multiply the submatrices. Figure 7 shows a order 4 cube layed out in the plane using the CHiP architecture. The numbers in the boxes show the index of the matrix elements initially contained in that processor. We are assuming that the processors are numbered in row major order. The dotted boxes show cubes of order 2. These cubes, which generally have order $2(k-1)$ contain an $\frac{n}{2} \times \frac{n}{2}$ submatrix of both A and B. Note that these cubes are constructed by "removing" the edges of order k and $2k$, where and edge of order k connects processors that are $2^{(k-1)}$ distance apart.

To compute the 2×2 matrix product, all processors exchange values of B on the order $2k$ edge and values of A on the order k edge. After the exchange, each cube of order $2(k-1)$ contains 4 submatrices of size $\frac{n}{2} \times \frac{n}{2}$. This is all the data that is required for each cube of order $2(k-1)$ to compute its part of the 2×2 matrix product independantly. If the submatrix is not a single element, two matrix products of $\frac{n}{2} \times \frac{n}{2}$ matrices are required. These matrix products are done using the same algorithm. Matrix addition is done element by element. Because corresponding elements of the matrices are contained in the same processor, no communication is required.

To find the cost of this cube matrix multiply, $K(CMM)$, we need to find the edge with the most messages. At the first level of recursion, the order k and $2k$ edges were used to send a message each way. This is the only use of these edges in the algorithm. Therefore, $w(e) = 1$, where e is a order k or $2k$ edge. At the second level of recursion, two matrix products are computed using the order $k-1$ and $2k-1$ edges. Each matrix product sends one message each way on each edge giving $w(e) = 2$, where e is a order $k-1$ or $2k-1$ edge. At level l of the recursion, $w(e) = 2^{l-1}$ messages over the order $k-(l-1)$ and $2k-(l-1)$ edges. The recursion stops when we have order 2 cubes. This is at the $\log n$ level of recursion. There are $\frac{n}{2}$ matrix multiplies done by order 2 cubes. These

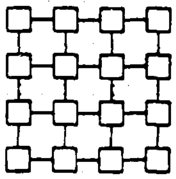


Figure 6: Another contraction of 16 logical processes to 4 processors.

WAP matrix multiply: ticks for n (items) on p (processors)				
Contraction	16 on 4	64 on 16	64 on 4	256 on 16
M_1	48854	111478	400452	901543
M_2	31113	73088	221545	7 7646

Table 2: Timings of the WAP matrix multiply algorithm.

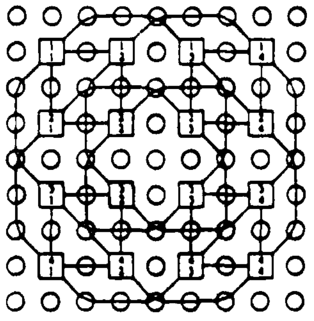


Figure 7: An order 4 binary n-cube.

order 2 cubes use the order 1 and $k+1$ edges. Each matrix multiply sends 1 message each way giving $w(e)=\frac{n}{2}$, where e is a order 1 or $k+1$ edge. Since this is the largest value, $K(CMM)=\frac{n}{2}$.

Consider any contraction, $M(CMM,p)$ where $p=2^m$ for some $m \le 2k$. $M(CMM,p)$ will map $\frac{n^2}{p}$ logical processes to every processor. This allows us to put a cube of order $\log\left(\frac{n^2}{p}\right)=2k-m$ into each processor. The processor-to-processor connection graph is also a cube and is of order m. Each processor-to-processor connection supports $\frac{n^2}{p}$ communication paths in the original graph. The real question is which sub-cube do we map to each processor. The cost of the contraction, $K(M(CMM,p))$ will be $\frac{n^2}{p}$ times the maximum $w(e)$, where e is mapped to a physical edge. If e is order 1 or $2k+1$ from the original cube, $K(M(CMM,p))=\frac{n^3}{2p}$.

Consider the contraction that maps the edges of order 1 through $\left\lceil\frac{2k-m}{2}\right\rceil$ and order $k+1$ through $k+\left\lceil\frac{2k-m}{2}\right\rceil$ into internal edges. This makes the edge of order $k+\left\lfloor\frac{2k-m}{2}\right\rfloor+1$ the edge with the most messages. This edge is used by level $k-\left\lfloor\frac{2k-m}{2}\right\rfloor$ of the recursion. From before we know that $w(e)=2^{k-\left\lfloor\frac{2k-m}{2}\right\rfloor-1}=\frac{\sqrt{p}}{2}$. Therefore $K(M(CMM,p))=\frac{n^2\sqrt{p}}{2p}$. Clearly, this contraction is better in terms of the number of messages over the busiest physical edge than any contraction that does not keep the high traffic logical edges internal to a processor.

By contrast, let us consider the Batcher bitonic merge sort. This sort runs on a order k cube to sort $n=2^k$ elements. The final sorting will have the smallest element in the first processor and the largest element in the last processor. Figure 8 shows a graphical representation of the algorithm. The arrows represent a data exchange and a compare, leaving the larger number at the end with the arrow and the smaller at the other end. It is obvious from the figure that the order 1 edge has the most messages. Therefore, $K(SORT)=\log n$.

Again, to contract this algorithm, we see that we want to assign a sub-cube into a processor. Consider the contraction $M(SORT,p)$ where the edges of order 1 through order $\log p$ are mapped to internal edges. We are assuming that $p=2^m$, for some $m \le \log n$. This contraction assigns the busiest logical edges to be internal edges. These edges carry $\log n - \log p$ messages. Since each processor contains $\frac{n}{p}$ logical processors, $K(M(SORT,p))=\frac{n(\log n - \log p)}{p}$. Any contraction that does not map these first $\log p$ edges to internal edges will have a higher communication cost. These results agree with and explain the results of Hsiao[7], even though his final algorithm was embedded in a grid instead of another cube.

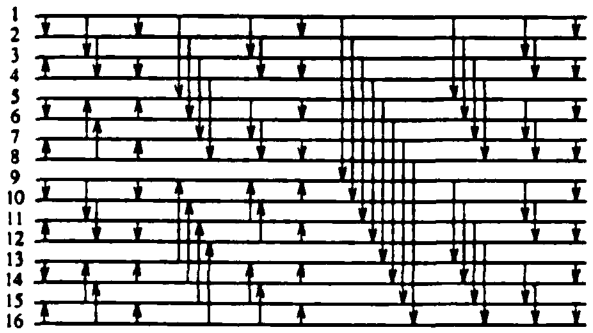


Figure 8: Batcher's bitonic merge sort.

In comparing the contractions for matrix multiply and Batcher's sort we see that the same size cube is mapped in a different way wher mapped to the same number of processors. The busiest edges are different for the two algorithms, thus, the contractions are different.

Conclusion

The algorithm contraction problem is an important problem for parallel programmers. The way in which an algorithm is contracted can have a significant affect on performance. Processor-to-processor communication can be used as a lower bound on the execution time for an algorithm. It is the processor-to-processor communication that is affected by different contractions.

We have looked at algorithms for the tree, grid, and binary n-cube interconnections. For each algorithm we have compared possible contractions of these algorithms. For trees, we proved that Leiserson's layout technique was the best for contracting tree algorithms such as minimum and sums. For grid algorithms, we conjectured that coalescing by maximizing the area for a given perimeter is optimal for the algorithms with balanced edge loadings. Finally, we showed two algorithms for binary n-cubes that required different contractions to produce the optimal results for the algorithm.

References

[1] L.M. Adams, *Iterative Algorithms for Large Sparse Linear Systems on Parallel Computers*, Ph.D. Thesis, University of Virginia, Charlottesville, November 1982.

[2] K.E. Batcher, "Sorting Networks and their Applications", *Proceedings of the AFIPS Sprint Joint Computer Confrence*, Vol 32, 1968, pp. 307-314.

[3] F. Berman, M. Goodrich, C. Koelbel, W.J. Robison III, K. Showell, "Prep-P: A Mapping Preprocessor for CHiP Architectures", *Proceedings of the 1985 International Confrence on Parallel Processing*, pp. 731-733.

[4] F. Berman, L. Snyder, "On Mapping Parallel Algorithms into Parallel Architectures", *Proceedings of the 1984 International Conference on Parallel Processing*, pp. 307-309.

[5] S.H. Bokhari, "On the Mapping Problem, *IEEE Transactions on Computers*, C-30, No. 3, March 1981, pp. 207-214.

[6] D. Gannon, L. Snyder, J. Van Rosendale, "Programming Substructure Computations for Elliptic Problems on the CHiP System", Ahmed K. Noor (ed.), *Impact of New Computing Systems of Computational Mechanics*, The American Society of Mechanical Engineers, 1983, pp. 65-80.

[7] C.C. Hsiao, *Highly Parallel Processing of Relational Databases*, Ph.D. Thesis, Purdue University, Department of Computer Sciences, December 1982.

[8] S.Y. Kung, K.S. Arun, R.J. Gal-Ezer, D.B. Bhaskr Rao, "Wavefront Array Processor: Language, Architecture, and Applications", *IEEE Transactions on Computers*, C-31, No. 11, November 1982, pp. 1-54-1065

[9] C.E. Leiserson, *Area-Efficient VLSI Computation*, MIT Press, Cambridge, Massachusetts, 1983.

[10] P.A. Nelson, *A Non-systolic Matrix Product Algorithm*, University of Washington, Department of Computer Science, Technical Report No. 85-11-02, November 1985.

[11] L. Snyder, "Introduction to the Configurable, Highly Parallel Computer", *Computer*, 15(1), January 1982, pp. 47-56.

[12] L. Snyder, "Parallel Programming and the Poker Programming Environment", *Computer*, 17(7), July 1984, pp. 27-36.

[13] L. Snyder, "Type Architectures, Shared Memory and the Corollary of Modest Potential", *Annual Review of Computer Science*, 1986 (to appear).